1. **Naive**
    - All forecasts for the future are equal to the last observed value of the series.
    - Assumes that the most recent observation is the only important one, and all previous observations provide no information for the future. This can be thought of as a weighted average where all of the weight is given to the last observation.

2. **Autoregression (AR)**

    Description:
    - The autoregression (AR) method models the next step in the sequence as a linear function of the observations at prior time steps.
    - The notation for the model involves specifying the order of the model p as a parameter to the AR function, e.g. AR(p). For example, AR(1) is a first-order autoregression model.
    - The method is suitable for univariate time series without trend and seasonal components.

    Software:
    - AR can be implemented in Python via the AutoReg Statsmodels package
      For example:

```
# AR example
from statsmodels.tsa.ar_model import AutoReg
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = AutoReg(data, lags=1)
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

    API and more information:
    - statsmodels.tsa.ar_model.AutoReg
      https://www.statsmodels.org/stable/generated/statsmodels.tsa.ar_model.AutoReg.html
    - statsmodels.tsa.ar_model.AutoRegResults
      https://www.statsmodels.org/stable/generated/statsmodels.tsa.ar_model.AutoRegResults.html
    - Autoregressive model https://en.wikipedia.org/wiki/Autoregressive_model

3. **Moving average**

Description:

- The moving average (MA) method models the next step in the sequence as a linear function of the residual errors from a mean process at prior time steps.
- A moving average model is different from calculating the moving average of the time series.
- The notation for the model involves specifying the order of the model q as a parameter to the MA function, e.g. MA(q). For example, MA(1) is a first-order moving average model.
- The method is suitable for univariate time series without trend and seasonal components.

Software:

- We can use the ARIMA class to create an MA model and setting a zeroth-order AR model. We must specify the order of the MA model in the order argument.

```
# MA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(0, 0, 1))
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

More information:

- Moving-average model on Wikipedia https://en.wikipedia.org/wiki/Moving-average_model

4. **Autoregressive Moving Average (ARMA)**

Description:

- The Autoregressive Moving Average (ARMA) method models the next step in the sequence as a linear function of the observations and residual errors at prior time steps.
- It combines both Autoregression (AR) and Moving Average (MA) models.
- The notation for the model involves specifying the order for the AR(p) and MA(q) models as parameters to an ARMA function, e.g. ARMA(p, q). An ARIMA model can be used to develop AR or MA models.
- The method is suitable for univariate time series without trend and seasonal components.

Software

- ARMA can be implemented in Python via the ARIMA package

```
# ARMA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(2, 0, 1))
model_fit = model.fit()
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

More information:

- Autoregressive–moving-average model https://en.wikipedia.org/wiki/Autoregressive%E2%80%93moving-average_model

5. **Autoregressive Integrated Moving Average (ARIMA)**

Description:

- The Autoregressive Integrated Moving Average (ARIMA) method models the next step in the sequence as a linear function of the differenced observations and residual errors at prior time steps.
- It combines both Autoregression (AR) and Moving Average (MA) models as well as a differencing pre-processing step of the sequence to make the sequence stationary, called integration (I).
- The notation for the model involves specifying the order for the AR(p), I(d), and MA(q) models as parameters to an ARIMA function, e.g. ARIMA(p, d, q). An ARIMA model can also be used to develop AR, MA, and ARMA models.
- The method is suitable for univariate time series with trend and without seasonal components.

Software:

```
# ARIMA example
from statsmodels.tsa.arima.model import ARIMA
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = ARIMA(data, order=(1, 1, 1))
model_fit = model.fit()
# make prediction
```

```
yhat = model_fit.predict(len(data), len(data), typ='levels')
print(yhat)
```

More information:

- Autoregressive integrated moving average
  https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

6. **Seasonal Autoregressive Integrated Moving-Average (SARIMA)**
   Description:
   - It combines the ARIMA model with the ability to perform the same autoregression, differencing, and moving average modeling at the seasonal level.
   - The notation for the model involves specifying the order for the AR(p), I(d), and MA(q) models as parameters to an ARIMA function and AR(P), I(D), MA(Q) and m parameters at the seasonal level, e.g. SARIMA(p, d, q)(P, D, Q)m where “m” is the number of time steps in each season (the seasonal period). A SARIMA model can be used to develop AR, MA, ARMA and ARIMA models.
   - The method is suitable for univariate time series with trend and/or seasonal components.

   Software:
   - SARIMA can be implemented in Python using the SARIMA package

```
# SARIMA example
from statsmodels.tsa.statespace.sarimax import SARIMAX
from random import random
# contrived dataset
data = [x + random() for x in range(1, 100)]
# fit model
model = SARIMAX(data, order=(1, 1, 1), seasonal_order=(0, 0, 0, 0))
model_fit = model.fit(disp=False)
# make prediction
yhat = model_fit.predict(len(data), len(data))
print(yhat)
```

API and more information:

- statsmodels.tsa.statespace.sarimax.SARIMAX
  https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html
- statsmodels.tsa.statespace.sarimax.SARIMAXResults
  https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAXResults.html

7. **Simple Exponential Smoothing**

Description:

- Suitable for forecasting univariate data with no clear trend or seasonal pattern.
- Attach larger weights to more recent observations than to observations from the distant past. Forecasts are calculated using weighted averages, where the weights decrease exponentially as observations come from further in the past - the smallest weights are associated with the oldest observations.
- It requires a single parameter, called alpha (a), also called the smoothing factor or smoothing coefficient. This parameter controls the rate at which the influence of the observations at prior time steps decay exponentially. Alpha is often set to a value between 0 and 1. Large values mean that the model pays attention mainly to the most recent past observations, whereas smaller values mean more of the history is taken into account when making a prediction.

Software:

- Simple Exponential Smoothing can be implemented in Python via the SimpleExpSmoothing Statsmodels class.
- First, an instance of the SimpleExpSmoothing class must be instantiated and passed the training data. The fit() function is then called providing the fit configuration, specifically the alpha value called smoothing_level. If this is not provided or set to None, the model will automatically optimize the value.
- This fit() function returns an instance of the HoltWintersResults class that contains the learned coefficients. The forecast() or the predict() function on the result object can be called to make a forecast.
  For example:

```
# single exponential smoothing
...
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
# prepare data
data = ...
# create class
model = SimpleExpSmoothing(data)
# fit model
model_fit = model.fit(...)
# make prediction
yhat = model_fit.predict(...)
```

API:

- statsmodels.tsa.holtwinters.SimpleExpSmoothing
  https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.SimpleExpSmoothing.html

8. **Double Exponential Smoothing**

Description:

- Double Exponential Smoothing is an extension to Exponential Smoothing that explicitly adds support for trends in the univariate time series.
- In addition to the alpha parameter for controlling smoothing factor for the level, an additional smoothing factor is added to control the decay of the influence of the change in trend called beta (b).
- The method supports trends that change in different ways: an additive and a multiplicative, depending on whether the trend is linear or exponential respectively.
- Double Exponential Smoothing with an additive trend is classically referred to as Holt's linear trend model, named for the developer of the method Charles Holt.

- Additive Trend: Double Exponential Smoothing with a linear trend.
- Multiplicative Trend: Double Exponential Smoothing with an exponential trend.

- For longer range (multi-step) forecasts, the trend may continue on unrealistically. As such, it can be useful to dampen the trend over time. Dampening means reducing the size of the trend over future time steps down to a straight line (no trend).
- As with modeling the trend itself, we can use the same principles in dampening the trend, specifically additively or multiplicatively for a linear or exponential dampening effect. A damping coefficient Phi (p) is used to control the rate of dampening.
- Hyperparameters:
    - Alpha: Smoothing factor for the level.
    - Beta: Smoothing factor for the trend.
    - Trend Type: Additive or multiplicative.
    - Dampen Type: Additive or multiplicative.
    - Phi: Damping coefficient.

Software:

- Double Exponential Smoothing can be implemented in Python using the ExponentialSmoothing Statsmodels class.
- First, an instance of the ExponentialSmoothing class must be instantiated, specifying both the training data and some configuration for the model.
- Specifically, you must specify the following configuration parameters:
    - trend: The type of trend component, as either "add" for additive or "mul" for multiplicative. Modeling the trend can be disabled by setting it to None.
    - damped: Whether or not the trend component should be damped, either True or False.
    - seasonal: The type of seasonal component, as either "add" for additive or "mul" for multiplicative. Modeling the seasonal component can be disabled by setting it to None.

        - seasonal_periods: The number of time steps in a seasonal period, e.g. 12 for 12 months in a yearly seasonal structure

- The model can then be fit on the training data by calling the fit() function.
- This function allows you to either specify the smoothing coefficients of the exponential smoothing model or have them optimized. By default, they are optimized (e.g. optimized=True). These coefficients include:
    - smoothing_level (alpha): the smoothing coefficient for the level.
    - smoothing_slope (beta): the smoothing coefficient for the trend.
    - smoothing_seasonal (gamma): the smoothing coefficient for the seasonal component.
    - damping_slope (phi): the coefficient for the damped trend.

- Additionally, the fit function can perform basic data preparation prior to modeling; specifically:
    - use_boxcox: Whether or not to perform a power transform of the series (True/False) or specify the lambda for the transform.
- The fit() function will return an instance of the HoltWintersResults class that contains the learned coefficients. The forecast() or the predict() function on the result object can be called to make a forecast.

```
# double or triple exponential smoothing
...
from statsmodels.tsa.holtwinters import ExponentialSmoothing
# prepare data
data = ...
# create class
model = ExponentialSmoothing(data, ...)
# fit model
model_fit = model.fit(...)
# make prediction
yhat = model_fit.predict(...)
```

API:

- statsmodels.tsa.holtwinters.ExponentialSmoothing https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html
- statsmodels.tsa.holtwinters.HoltWintersResults https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.HoltWintersResults.html
- forecast: Forecasting Functions for Time Series and Linear Models R package https://cran.r-project.org/web/packages/forecast/index.html

Machine Learning models for time series forecasting:

1. Multi-Layer Perceptron (MLP)
2. Bayesian Neural Network (BNN)
3. Radial Basis Functions (RBF)
4. Generalized Regression Neural Networks (GRNN), also called as kernel regression
5. K-Nearest Neighbor regression (KNN)
6. CART regression trees (CART)
7. Support Vector Regression
8. Gaussian Processes
9. Recurrent Neural Networks
10. Long Short-Term Memory (LSTM)

Sources:

1. Statistical and Machine Learning forecasting methods: Concerns and ways forward https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0194889
2. Exponential smoothing https://onlinelibrary.wiley.com/doi/abs/10.1002/for.3980040103
3. Simple Exponential Smoothing https://otexts.com/fpp2/ses.html
4. A Gentle Introduction to Exponential Smoothing for Time Series Forecasting in Python https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/#:~:text=Single%20Exponential%20Smoothing%2C%20SES%20for,smoothing%20factor%20or%20smoothing%20coefficient.
5. Forecasting: Principles and Practice https://otexts.com/fpp2/